

## Contenido

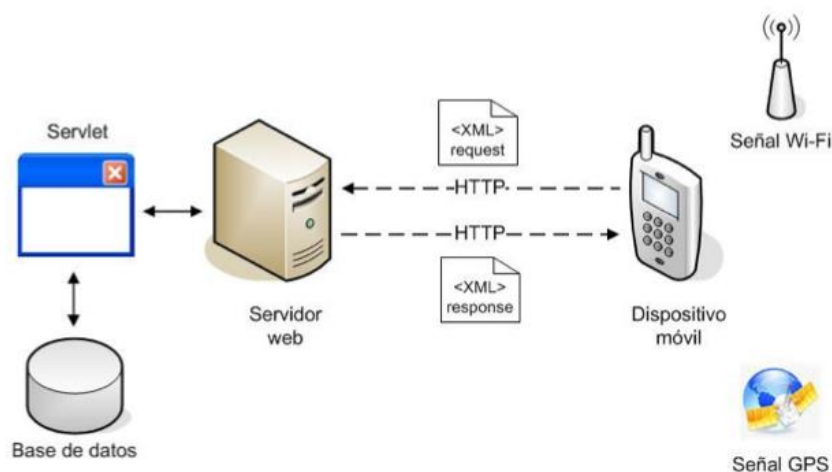
INTRODUCCIÓN. Acceso a bases de datos remotas .....	215
Acceso a ApiRest con Retrofit .....	217
<i>Definición del Servicio Rest</i> .....	218
<i>Consumo de un Servicio Rest desde Android</i> .....	219
Acceso a BD con FIREBASE.....	223
<i>Creando un app de Firebase</i> .....	223
<i>Filtrado y ordenación</i> .....	230
<i>Crear un proyecto con Autentificación</i> .....	233
<i>FirebaseUI y RecyclerView</i> .....	235
<i>Instalar servicios de google en Genimotion.</i> .....	240

# 12.

## Acceso a datos externas. ApiREST y FireBase

### INTRODUCCIÓN. Acceso a bases de datos remotas

Para obtener y guardar información de una base de datos remota, es necesario conectarse a un servidor donde se encontrará la BBDD. El esquema de funcionamiento lo podemos ver en la siguiente imagen:



En el servidor funcionan en realidad tres componentes básicos:

- Una base de datos, que almacena toda la información de los usuarios. Para la BBDD se puede utilizar MySQL, que es de licencia gratuita.



- Un *servlet*, que atiende la petición recibida, la procesa y envía la respuesta correspondiente. Un *servlet* no es más que un componente Java, generalmente pequeño e independiente de la plataforma.
- Un *servidor web*, donde reside y se ejecuta el *servlet*, y que permanece a la espera de conexiones HTTP entrantes. (usaremos Apache)

Los formatos más utilizados para compartir información mediante estos servicios *web* son *XML* (y otros derivados) y *JSON*.



¿Qué es JSON?

JSON (*Javascript Object Notation*) es un formato ligero de intercambio de datos entre clientes y servidores, basado en la sintaxis de *Javascript* para representar estructuras en forma organizada. Es un formato en texto plano independiente de todo lenguaje de programación, es más, soporta el intercambio de datos en gran variedad de lenguajes de programación como *PHP*, *Python*, *C++*, *C#*, *Java* y *Ruby*.

*XML* también puede usarse para el intercambio, pero debido a que su definición genera un *DOM*, el parseo se vuelve extenso y pesado. Además de ello *XML* debe usar *Xpath* para especificar rutas de elementos y atributos, por lo que demora la reconstrucción de la petición. En cambio *JSON* no requiere restricciones adicionales, simplemente se obtiene el texto plano y el engine de *Javascript* en los navegadores hace el trabajo de parsing sin ninguna complicación.

### **Tipos de datos en JSON**

Similar a la estructuración de datos primitivos y complejos en los lenguajes de programación, *JSON* establece varios tipos de datos: cadenas, números, booleanos, arrays, objetos y valores null.

El propósito es crear objetos que contengan varios atributos compuestos como pares clave-valor. Donde la clave es un nombre que identifique el uso del valor que lo acompaña.

Veamos un ejemplo:

```
{
  "Id": 101,
  "Nombre": "Carlos",
  "EstaActivo": true,
  "Notas": [ 2.3, 4.3, 5.0]
}
```

La anterior estructura es un objeto *JSON* compuesto por los datos de un estudiante. Los objetos *JSON* contienen sus atributos entre llaves “{}”, al igual que un bloque de código en *Javascript*, donde cada atributo debe ir separado por coma ‘,’ para diferenciar cada par.

La sintaxis de los pares debe contener dos puntos ‘:’ para dividir la clave del valor. El nombre del par debe tratarse como cadena y añadirle comillas dobles.

Si notas, este ejemplo trae un ejemplo de cada tipo de dato:

- El *Id* es de tipo entero, ya que contiene un número que representa el código del estudiante.
- El *Nombre* es un string. Usa comillas dobles para identificarlas.

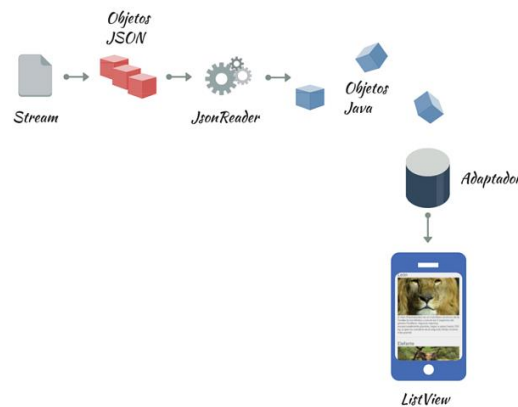


- `EstaActivo` es un tipo booleano que representa si el estudiante se encuentra en la institución educativa o no. Usa las palabras reservadas `true` y `false` para declarar el valor.
- `Notas` es un arreglo de números reales. El conjunto de sus elementos debes incluirlos dentro de corchetes “[ ]” y separarlos por coma.

La idea es crear un mecanismo que permita recibir la información que contiene la base de datos externa en formato JSON hacia la aplicación. Con ello se parseará cada elemento y será interpretado en forma de objeto Java para integrar correctamente el aspecto en la interfaz de usuario.

La clase **`JsonObject`** de la librería **`org.json.JSONObject`**, puede interpretar datos con formato JSON y ‘parsearlos’ a objetos Java o a la inversa.

Veamos una ilustración que muestra el proceso de parseo que será estudiado:



- Como puedes observar el origen de los datos es un servidor externo o hosting que hayas contratado como proveedor para tus servicios web. La aplicación web que realiza la gestión de encriptación de los datos a formato JSON puede ser PHP, JavaScript, ASP.NET, etc..
- Tu aplicación Android a través de un cliente realiza una petición a la dirección URL del recurso con el fin de obtener los datos. Ese flujo entrante debe interpretarse con ayuda de un parser personalizado que implementaran las clases que se utilizan para trabajar con JSON.
- El resultado final es un conjunto de datos adaptable al API de Android. Dependiendo de tus necesidades, puedes convertirlos en una lista de objetos estructurados que alimenten un adaptador que pueble un `ListView` o simplemente actualizar la base de datos local de tu aplicación en `SQLite`.

## Acceso a ApiRest con Retrofit

Retrofit es un cliente REST para Android y Java, desarrollada por Square, muy simple y fácil de aprender. Permite hacer peticiones GET, POST, PUT, PATCH, DELETE y HEAD; gestionar diferentes tipos de parámetros y parsear automáticamente la respuesta a un POJO (Plain Old Java Object).



Vamos a utilizar esta librería por su facilidad de manejo. Pero antes, aunque no es propio de esta asignatura, nos crearemos un sencillo ApiRest para poder acceder desde nuestra aplicación.

## Definición del Servicio Rest

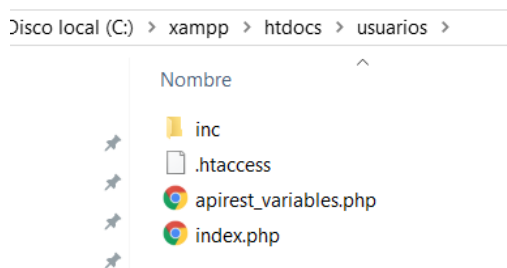
El siguiente paso es crear nuestro servicio. Para crear el servicio vamos a utilizar el marco WEB API REST, este nos facilitará su construcción. La arquitectura REST (*Representational State Transfer*) trabaja sobre el protocolo HTTP. Por consiguiente, los procedimientos o métodos de comunicación son los mismos que HTTP, siendo los principales: GET, POST, PUT, DELETE. Otros métodos que se utilizan en RESTful API son OPTIONS y HEAD. Este último se emplea para pasar parámetros de validación, autorización y tipo de procesamiento, entre otras funciones.

Otro componente de un RESTful API es el “HTTP Status Code”, que le informa al cliente o consumidor del API que debe hacer con la respuesta recibida. Estos son una referencia universal de resultado, es decir, al momento de diseñar un RESTful API toma en cuenta utilizar el “Status Code” de forma correcta. Los códigos de estado, más utilizados son:

200 OK	
201 Created (Creado)	304 Not Modified (No modificado)
400 Bad Request (Error de consulta)	401 Unauthorized (No autorizado)
403 Forbidden (Prohibido)	404 Not Found (No encontrado)
500 Internal Server Error (Error Interno de Servidor)	422 (Unprocessable Entity (Entidad no procesable))

Hasta que alojemos nuestro servicio Rest en un hosting, tendremos que proporcionar nosotros los mecanismos adecuados que gestionen la BD con MySQL, y el funcionamiento de PHP. Para este último se deberá soportar 5.5 o superior. Para ello podremos utilizar cualquier herramienta que conozcamos, como puede ser Wamp Server o Xamp. Nosotros para nuestro ejemplo usaremos el servidor Xampp descargado de forma gratuita de <https://www.apachefriends.org>

Usaremos el mismo ApiRest que ya habéis usado en el módulo de Acceso a Datos, para lo cual solo tendremos que copiar la carpeta que contiene la implementación, dentro de htdocs del servidor Xampp y dentro de la carpeta que hayáis creado para vuestro proyecto.



La configuración de nuestro API consta de dos archivos:

- a) **.htaccess** En nuestro archivo .htaccess configuraremos las reglas de acceso, sobre una ruta que le indiquemos en RewriteBase (aquí es donde tendremos que añadir la carpeta en la que habremos alojado nuestra API, comenzando y acabando con /).

```
RewriteEngine On
RewriteBase "/usuarios/"
RewriteRule ^([a-zA-Z_-]*)$ index.php?action=$1%{QUERY_STRING}
RewriteRule ^([a-zA-Z_-]*)/([a-zA-Z0-9_-]*)$ index.php?action=$1&value=$2%{Q
RewriteRule ^([a-zA-Z_-]*)/([a-zA-Z_-]*)/([a-zA-Z0-9_-]*)$ index.php?action=$1
```

- b) **apiest\_variables.php** definiremos los datos de conexión a la base de datos. He indicaremos las tablas que tiene esta y el nombre del identificador de cada una de ellas.

```
<?php

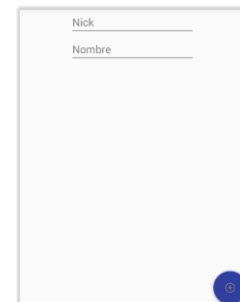
// CONFIGURACIÓN BASE DE DATOS MYSQL
$servername = "127.0.0.1";
$username = "root";
$password = "";

// BASE DE DATOS
$dbname = "usuariosmensajes";

// TABLAS Y SU CLAVE
$tablas = array();
$tablas["mensajes"]="_id";
$tablas["usuarios"]="_id";
```

El resto de archivos del ApiREST no tendrán que modificarse, Ya que está construida de forma genérica con las necesidades más comunes para estos casos.

Vamos a suponer un ejemplo muy sencillo de una base de datos Usuarios en el que tendremos solamente una tabla Usuarios con dos campos de tipo String (nick y nombre). La BD la habremos construido en el servidor con antelación (en este caso con tabla usuarios de tres campos, nick y nombre de tipo cadena y \_id de tipo numérico autoincrementable como clave). También crearemos una aplicación con dos campos de texto que nos permita insertar los datos del usuario y un botón flotante de añadir, como se ve en la imagen siguiente:

A screenshot of a web form for user registration. It has two input fields, one labeled 'Nick' and one labeled 'Nombre'. At the bottom right, there is a blue circular button with a white plus sign, indicating an 'add' or 'submit' function.

## Consumo de un Servicio Rest desde Android

Existen diferentes librerías que nos permitirían consumir los servicios desde la App de Android, pero dada su facilidad vamos a utilizar las librerías: Retrofit y Gson.

Retrofit la utilizaremos para hacer peticiones y procesar las respuestas del APIRest, mientras que con Gson transformaremos los datos de JSON a los propios que utilice la aplicación.

Para ello añadiremos las siguientes líneas en el build.gradle de la app, y no olvides incluir permisos de internet:

```
implementation 'com.squareup.retrofit2:retrofit:2.7.0'
implementation 'com.squareup.retrofit2:converter-gson:2.7.0'
```

Podemos decir que los pasos a seguir serán los siguientes:



1. Creación de un builder de Retrofit. Para poder utilizar Retrofit necesitas crear un builder que te servirá para comunicarte con la API elegida. En la imagen se ha utilizado un método para encapsular el código, aunque no es necesario.

```
private ProveedorServicios crearRetrofit() {  
    String url = "http://10.0.2.2/usuarios/"; //para el AVD de android  
    //"http://xusa.iesdoctorbalmis.info/usuarios/"; //para servidor del instituto  
    Retrofit retrofit = new Retrofit.Builder()  
        .baseUrl(url)  
        .addConverterFactory(GsonConverterFactory.create())  
        .build();  
  
    return retrofit.create(ProveedorServicios.class);  
}
```

Si te fijas en la segunda línea verás que hay que escribir la url base de la API a la que harás las peticiones. La obtienes de la documentación proporcionada por el creador de la API.

En la siguiente línea hay que indicar la forma de convertir los objetos de la API a los de tu aplicación y viceversa, aquí es donde especificas que vas a utilizar Gson.

Este builder lo necesitarás para hacer las llamadas a la API, así que procura que sea accesible

2. Creación de las clases Pojo que le servirán al Gson para parsear sus resultados. Pero primero necesitas conocer la estructura del Json que va a devolverte la API, ya que no hay un patrón establecido. Para conocer la estructura previamente, se puede utilizar PostMan y realizar las diferentes peticiones (GET, POST, PUT, etc) desde este. Existen aplicaciones o webs que facilitan la creación de la clase Pojo a partir de un JSon, como por ejemplo: <http://www.jsonschema2pojo.org/>

```
class RespuestaJSON {  
    public int respuesta;  
    public String metodo;  
    public String tabla;  
    public String mensaje;  
    public String sqlQuery;  
    public String sqlError;  
}
```

Después tienes que crear la estructura de clases para almacenar la información que te resulte útil. Clase Usuario en este ejemplo.

```
public class Usuarios {  
    String nick, nombre;  
  
    public Usuarios(String nick, String nombre) {  
        this.nick = nick;  
        this.nombre = nombre;  
    }  
  
    public String getNick() {  
        return nick;  
    }  
}
```

3. Otro elemento imprescindible, es la gestión de los servicios que se quieran utilizar. Para cada uno de ellos se tendrá que hacer una petición a la API. Necesitaremos crear una interfaz con todos los servicios que quieras utilizar. Aquí tienes unos ejemplos:



```

public interface ProveedorServicios {
    @GET("usuarios")
    @Headers({"Accept: application/json", "Content-Type: application/json"})
    Call<List<Usuarios>> getUsuarios();

    @GET("mensajes")
    @Headers({"Accept: application/json", "Content-Type: application/json"})
    Call<List<Mensaje>> getMensajes();

    @GET("mensajes/{nick}")
    @Headers({"Accept: application/json", "Content-Type: application/json"})
    Call<List<Usuarios>> getUsuario(@Path("nick") String nick);

    @POST("usuarios")
    @Headers({"Accept: application/json", "Content-Type: application/json"})
    Call<RespuestaJson> insertarUsuario(@Body Usuarios usuarios);

    @POST("mensajes")
    @Headers({"Accept: application/json", "Content-Type: application/json"})
    Call<RespuestaJson> insertarMensaje(@Body Mensaje mensaje);
}

```

En el código de arriba hay cinco servicios dos son de tipo GET sin palabra clave y que servirán para obtener todos los usuarios y todos los mensajes. Luego se tiene otro servicio GET con palabra clave (Nick) que servirá para seleccionar los mensajes de un determinado Nick. Y luego dos POST a los que se les pasa en el Body los datos a añadir.

Usando Call se realizará la llamada a la API, además ahí es donde se indica el tipo de dato que esperas obtener, en estos ejemplos la respuesta puede ser o una lista del tipo de objeto de la petición o un tipo RespuestaJson, que tendrá todos los posibles miembros que nos podría dar alguna de las llamadas invocadas.

Las palabras seguidas del símbolo '@' dan funcionalidad a los servicios, haciéndolos dinámicos y reutilizables, para más información <http://square.github.io/retrofit/>

- las que gestiona Retrofit para invocar la url de la petición son **@GET**, **@POST**, **@PUT** y **@DELETE**. El parámetro corresponderá con la url de la petición.

En el builder de Retrofit se incluye la url base del api terminada con /, que unida con el parámetro del servicio crearán la url completa de la petición:

**@GET** → `http://10.0.2.2/usuarios/` + `usuario`

- **@Header** y **@Headers**. Se usan para especificar los valores que vayan en la sección "header" de la petición, como por ejemplo en que formato van a ser enviados y recibidos los datos.
- **@Path**. Sirve para incluir un identificador en la url de la petición, para obtener información sobre algo específico. El atributo en el método de llamada que sea precedido por @Path sustituirá al identificador entre llaves de la ruta, que tenga el mismo nombre.
- **@Fields**. Nos permite pasar variables básicas en las peticiones Post y Put.
- **@Body**. Es equivalente a Fields pero para variables objeto.
- **@Query**. Se usa cuando la petición va a necesitar parámetros (los valores que van después del '?' en una url).





4. Pedir datos a la Api sería el último paso a realizar. Retrofit nos da la opción de realizarlo de manera síncrona o asíncrona. Si eligiéramos la manera síncrona deberíamos crear nuestro propio hilo para gestionar el acceso a la petición. Así que nosotros utilizaremos la forma asíncrona, en la cual se encargar la librería de la gestión, facilitando mucho el código.

```
private void anyadirUsuario(Usuarios usuarios)
{
    ProveedorServicios proveedorServicios = crearRetrofit();
    Call<RespuestaJSON> responseCall=proveedorServicios.insertarUsuario(usuarios);
    //Llamada asincrona gestionada por Retrofit y para ahorrar hilos
    responseCall.enqueue(new Callback<RespuestaJSON>() {
        @Override
        public void onResponse(Call<RespuestaJSON> call, Response<RespuestaJSON> response) {
            RespuestaJSON usuariosResponse = response.body();
            if(usuariosResponse !=null) Toast.makeText(getActivity(), usuariosResponse.toString(), Toast.LENGTH_LONG).show();
            else Toast.makeText(getActivity(),response.message(), Toast.LENGTH_LONG).show();
            if( espera!=null)espera.dismiss();
            limpiaControl();
        }
        @Override
        public void onFailure(Call<RespuestaJSON> call, Throwable t) {
            Log.e("Error", t.toString());
            Toast.makeText(getActivity(), "Error" + t.toString(), Toast.LENGTH_LONG).show();
            if( espera!=null)espera.dismiss();
            limpiaControl();
        }
    });
}
```

A partir de una instancia del builder que hemos creado con anterioridad (llamada a crearRetrofit), crearemos un objeto de tipo Call, invocando al método adecuado del proveedor de servicios:

```
ProveedorServicios proveedorServicios = crearRetrofit();
Call<RespuestaJSON> responseCall=proveedorServicios.insertarUsuario(usuario);
```

Callback es una interfaz proporcionada por Retrofit. En ella puedes especificar tu objeto de respuesta, en este caso será un objeto RespuestaJSON. Con este objeto invocaremos a enqueue, de este modo la llamada será asíncrona.

```
responseCall.enqueue(new Callback<RespuestaJSON>())
```

Callback tiene dos métodos: void onResponse(Call call, Response response) y void onFailure(Call call, Throwable t). Cuando termine la petición según si ha tenido éxito o no llamará a los métodos onResponse u onFailure del callback respectivamente.

5. Ya sólo queda obtener la respuesta. Esto se hace en la clase en la que hayas implementado la interfaz Callback de Retrofit. Si todo ha ido bien, lo único que tienes que hacer es obtener el body del parámetro "response" del método onResponse, que coincide con el tipo de objeto que le pasaste y utilizar esta respuesta para la salida de tu aplicación.

```
RespuestaJSON usuariosResponse = response.body();
if(usuariosResponse !=null) Toast.makeText(getActivity(), usuariosResponse.message, Toast.LENGTH_LONG).show();
else Toast.makeText(getActivity(),response.message(), Toast.LENGTH_LONG).show();
```

 EjercicioPropuestoBDExternas

 Ejercicio Propuesto proyecto agenda



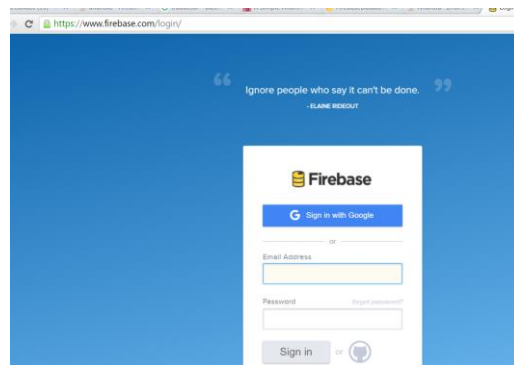
## Acceso a BD con FIREBASE

[Firebase](#) es una plataforma de backend para construir aplicaciones móviles y web, se encarga del manejo de la infraestructura permitiendo que el desarrollador se enfoque en otros aspectos de la aplicación. Entre sus características se incluye base de datos de tiempo real, autenticación de usuarios y almacenamiento (hosting) estático. La idea de usar Firebase es no tener que escribir código del lado del servidor y aprovechar al máximo las características que nos provee la plataforma.

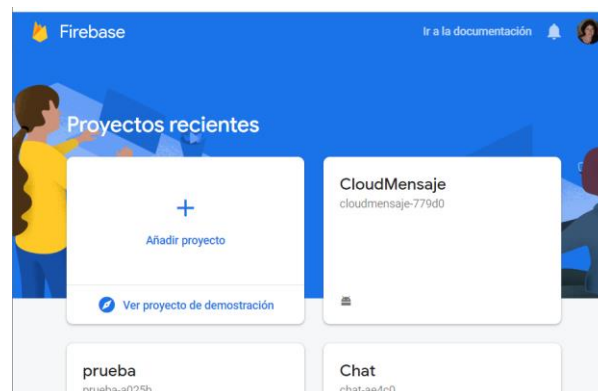
Para utilizar Firebase con Android disponemos de un SDK, lo que nos permitirá integrarlo fácilmente a nuestra aplicación. Para este ejemplo vamos a construir un listado de cosas por hacer usando la base de datos de tiempo real de Firebase como backend y autenticación con email/password y Twitter. El contenido se divide en tres partes, en esta primera parte de la guía haremos un app simple para probar enviar y recibir datos con Firebase.

### *Creando un app de Firebase*

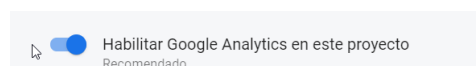
El primer paso es crear nuestra cuenta de Firebase, para ello le daremos a comenzar y crearemos nuestra cuenta con los datos que pidan:



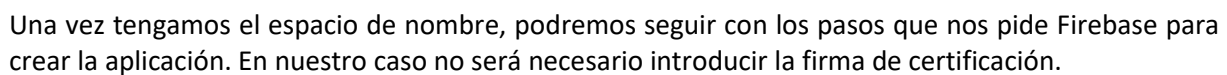
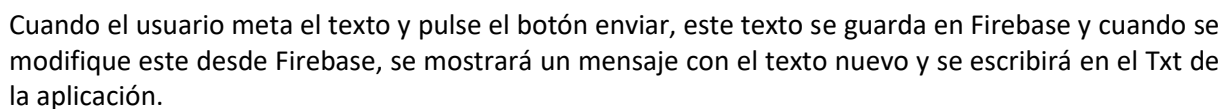
Una vez está lista veremos la pantalla donde es posible visualizar nuestros proyectos o crear nuevos.



Para crear un nuevo proyecto solamente tendremos que introducir un nombre y el lugar de origen. Indicaremos que lo que queremos es un proyecto de Android, posteriormente nos aparecerá la pantalla que nos permitirá añadir la aplicación a nuestro proyecto. Google nos pedirá si queremos habilitar Google Analytics en el proyecto, lo dejaremos habilitado.



Vamos a crear una aplicación sencilla con el siguiente aspecto, que nos permita entender el funcionamiento de acceso a Firebase para enviar y recibir datos sin autenticación.



Justo en el momento que termina de crearse la aplicación, se descargara un archivo JSON que deberemos copiar en nuestro proyecto Android. Solo tendremos que seguir las instrucciones que proporciona muy claramente la página Web de Firebase (copiar el archivo y añadir las líneas que indican que vamos a usar servicios de google en los build.gradle de la app y del proyecto).

224



## 1. En el proyecto

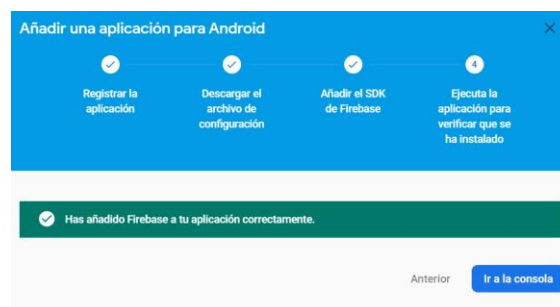
```
implementation 'com.google.firebase:firebase-analytics:17.2.0' //Analytic
implementation 'com.google.firebase:firebase-database:19.2.0' //Conexión a database
}
apply plugin: 'com.google.gms.google-services' //Servicios de google
```

## 2. En la APP

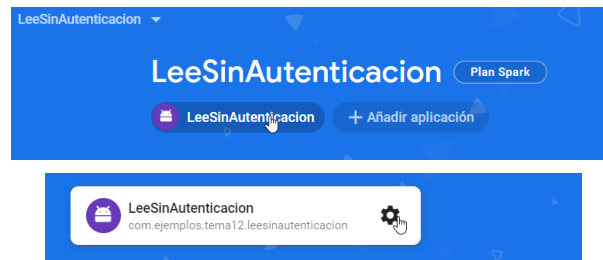
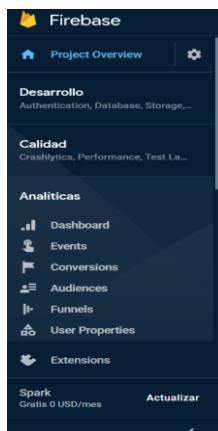
Muy importante no olvidar añadir en el SDK, el servicio de Google Play

<input type="checkbox"/> Google Play Licensing Library	1	Not installed
<input checked="" type="checkbox"/> Google Play services	49	Installed
<input checked="" type="checkbox"/> Google USB Driver	12	Not installed
<input type="checkbox"/> Google Web Driver	2	Not installed

Comprobará si nuestra aplicación se ha comunicado con sus servidores, para ello tendremos que ejecutarla y esperar para ver si es reconocida.



En el modo consola podremos acceder a todos nuestros proyectos, y podemos comprobar la aplicación o aplicaciones añadidas sobre un proyecto (podemos añadir más de una).



Accediendo a la configuración de la aplicación o del proyecto, podremos ver información sobre las claves, ID de aplicación y demás. También podremos descargar de nuevo el .json de configuración de los servicios.

Todas las aplicaciones por defecto se encuentran en modo desarrollo y bajo el plan gratis que soporta hasta 100 conexiones concurrentes, 1GB de almacenamiento y 10GB de transferencia en el backend.

Luego de creada el app nos dirigimos a ver sus detalles, podemos entrar en diversas pestañas que nos permitirán trabajar con la APP, pero a nosotros nos interesa la BD, por lo que haciendo click en el botón de Database, nos llevará a la pantalla de administración de la BD.

Firebase ofrece dos soluciones de bases de datos en la nube y accesibles para los clientes que admiten sincronización en tiempo real:

- **Cloud Firestore** es la base de datos más reciente de Firebase. Aprovecha lo mejor de Realtime Database con un modelo de datos nuevo, permite consultas más ricas y rápidas, y el escalamiento se ajusta a un nivel más alto que Realtime Database.

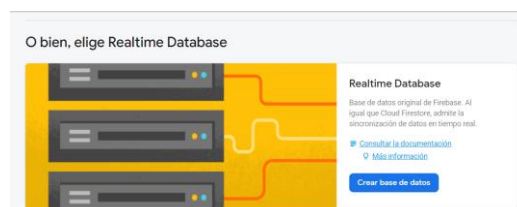


- **Realtime Database** es la base de datos original de Firebase. Es una solución eficiente y de baja latencia destinada a las apps para dispositivos móviles que necesitan estados sincronizados entre los clientes en tiempo real.

Realtime Database	Cloud Firestore
Almacena datos como un gran árbol JSON	Almacena datos como colecciones de documentos
<ul style="list-style-type: none"> <li>• Los datos simples son muy fáciles de almacenar.</li> <li>• Los datos complejos y jerárquicos son más difíciles de organizar a gran escala.</li> </ul>	<ul style="list-style-type: none"> <li>• Los datos simples son fáciles de almacenar en documentos, que son muy similares a JSON.</li> <li>• Los datos complejos y jerárquicos son más fáciles de organizar a escala, con subcolecciones dentro de los documentos.</li> <li>• Necesita menos desnormalización y compactación de datos.</li> </ul>

## RealtimeDatabase

Por defecto se crea una BD CloudFirestore, pero avanzaremos un poco más hasta llegar a la opción RealtimeDatabase, comenzaremos con esta ya que será más sencillo entender el concepto al estar acostumbrados a json.



Seleccionaremos la opción de comenzar en modo de prueba, para que cualquier usuario pueda acceder a escribir y leer en ella. Esto podrá ser modificado en las reglas de seguridad.



## Estructura de datos

Firebase almacena los datos en formato JSON, es decir, no es relacional por lo que es importante entender como se accede a los datos almacenados para construir un diseño eficiente. Es posible realizar hasta 32 niveles de datos anidados pero es una mala idea diseñar para anidar porque al obtener una referencia cualquiera, también se obtienen todos los nodos hijos, de allí que se sugiera tener una estructura lo más plana posible y se manejen índices.

En el caso de nuestra aplicación, por cada elemento vamos a guardar la descripción del ítem. Tendremos un solo listado con todos los ítems haremos uso de la función push() de Firebase para que genere un identificador único para cada elemento guardado de tal forma que no haya ningún conflicto.

En un principio nos generaremos directamente en la consola de Firebase, la rama ítems con un elemento, el resto de elementos los añadiremos desde nuestra aplicación.



[leesinautenticacion-243cd](#) > [items](#)

items

asdfggse

msg: "Hola mundo"

## Referencias

Es importante notar que de cada uno de los elementos hijos en la estructura de datos tiene su propio URL.

<https://leesinautenticacion-243cd.firebaseio.com/items/asdfggse/msg>

[leesinautenticacion-243cd](#) > [items](#) > [asdfggse](#) > [msg](#)

msg: "Hola mundo"

Cada Base de datos tendrá una URL asignada que coincide, en parte, con el nombre de la aplicación en el dominio firebaseIO.com que será la referencia que se para acceder a la BD. Además a cada uno de sus elementos hijos se puede acceder haciendo uso de la URL correspondiente. Al trabajar con Android, estas URLs pueden ser utilizadas como referencia para leer y escribir datos además de tener cada hijo un key asociado.

Un elemento importante y que todavía no hemos mencionado, son la Reglas. La Firebase Realtime Database proporciona un lenguaje de reglas flexibles basadas en expresiones y sintaxis similar a la de JavaScript, que permite definir fácilmente la manera en que tus datos deben estructurarse e indexarse, y el momento en que pueden someterse a lectura y escritura <https://firebase.google.com/docs/database/security>. Estas reglas las podremos configurar desde la pestaña RULES de nuestra BD, por defecto vendrían configuradas para usuarios autenticados (no es nuestro caso, ya que hemos elegido la opción modo de prueba al crear la BD).

```
{
  "rules": {
    ".read": "auth != null",
    ".write": "auth != null"
  }
}
```

Si nos hubiésemos confundido o quisiéramos cambiar las reglas, habría que cambiar su valor y sobre todo **No olvidar Publicar**.

```
{
  "rules": {
    ".read": true,
    ".write": true
  }
}
```

El siguiente paso será crear las líneas necesarias que nos permitan escribir y detectar cambios en la BD. Para ello tendremos que añadir la dependencia:

```
implementation 'com.google.firebase:firebase-database:19.2.0' //Conexión a database
```

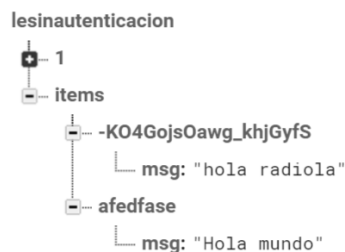
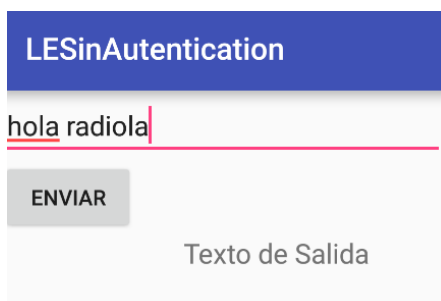


Solo nos quedará crear un objeto de tipo FirebaseDatabase, asociarle una instancia de FirebaseDatabase (esto relacionará nuestro objeto con los datos del archivo .json descargado anteriormente) y posteriormente usar los métodos necesarios para añadir los datos. El código lo podemos ver en la siguiente imagen:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    etTexto=(EditText)findViewById(R.id.etTexto);
    tvSalida=(TextView)findViewById(R.id.tvSalida);
    btEnviar=(Button)findViewById(R.id.btEnviar);
    final FirebaseDatabase database = FirebaseDatabase.getInstance();

    btEnviar.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            if (!etTexto.getText().toString().isEmpty())
                database.getReference("items").push().setValue(new Items(etTexto.getText().toString()));
        }
    });
}
```

En el onClick podemos ver el código que nos permite introducir el texto del EditText en el hijo ítems (que habíamos creado desde la consola), pero antes hemos creado una clave única con el método push(). Esto nos permitirá que ningún hijo de ítems tenga el valor repetido, aunque su valor interno se repita. Se puede ver el resultado en las imágenes siguientes:



Para insertar el valor del nuevo hijo hemos utilizado un objeto de la clase Pojo Items que nos habremos creado con anterioridad, Firebase convierte automáticamente los atributos con sus valores para poder guardarlos correctamente. Cuidado deberemos tener los atributos públicos o usar getter y setter.

```
class Items
{
    public String msg;

    public Items(String msg) { this.msg = msg; }

    @Override
    public String toString() { return "msg='" + msg + '\''; }
}
```

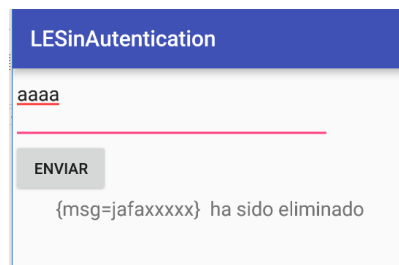
Si lo que queremos es controlar los cambios que ocurren en la BD, sea a través de una aplicación o directamente desde la consola de Firebase, tendremos que usar un listener. En nuestro caso nos



interesa utilizar `ChildEventListener` que provee los métodos: `onChildChanged`, `onChildAdded`, `onChildRemoved`, `onChildMoved`, `onCancelled` que controlan los distintos eventos de un cambio en los datos. Este objeto lo asignaremos al hijo ítems de la referencia de nuestra app a Firebase con una llamada a `addChildEventListener`. Dentro de este método, obtenemos el texto y lo mostramos en el `TextView`. Existen otros métodos que pueden servirnos de ayuda para realizar lo que necesitemos en cada momento, como por ejemplo el escuchador `addValueEventListener()`, que notificará de manera genérica si ha habido algún cambio en el nodo referenciado o en alguno de sus hijos (Mirar en la documentación de Firebase para más información).

```
childEventListener = database.getReference("items").addChildEventListener(new ChildEventListener() {  
    @Override  
    public void onChildAdded(DataSnapshot dataSnapshot, String s) {  
        tvSalida.setText(dataSnapshot.getValue().toString());  
    }  
    @Override  
    public void onChildChanged(DataSnapshot dataSnapshot, String s) {  
        Items valor = dataSnapshot.getValue(Items.class);  
        tvSalida.setText(valor.msg);  
    }  
    @Override  
    public void onChildRemoved(DataSnapshot dataSnapshot) {  
        tvSalida.setText(dataSnapshot.getValue().toString() + " ha sido eliminado");  
    }  
    @Override  
    public void onChildMoved(DataSnapshot dataSnapshot, String s) {  
    }  
    @Override  
    public void onCancelled(DatabaseError databaseError) {  
    }  
});
```

Cada vez que hacemos un cambio, se mostrará de la siguiente forma:



Un objeto `DataSnapshot` representa básicamente una rama del árbol JSON, es decir, contendrá toda la información de un nodo determinado, con su clave, su valor, y su listado de nodos hijos (que a su vez pueden tener otros descendientes), por el que podremos navegar libremente. Podremos obtener la clave y el valor mediante los métodos `getKey()` y `getValue()` respectivamente. Los subnodos los podremos obtener en su totalidad mediante `getChildren()` (los recibiremos en forma de listado de objetos `DataSnapshot`) o bien podremos navegar a subnodos concretos mediante `child("nombre-subnodo")`.

Otro tema importante a tener en cuenta es que la suscripción a una referencia de una base de datos de Firebase, es decir, el hecho de asignar un listener a una ubicación del árbol JSON para estar al tanto de sus cambios no es algo “gratuito” desde el punto de vista de consumo de recursos. Por tanto, es recomendable eliminar esa suscripción cuando ya no la necesitamos. Para hacer esto basta con llamar al método `removeEventListener()` sobre cada referencia a la base de datos que ya no sea necesaria y pasarle como parámetro el listener que deseamos eliminar.





```

@Override
protected void onDestroy() {
    if(database!=null && childEventListener !=null)
        database.getReference("items").removeEventListener(childEventListener);
    super.onDestroy();
}

```

Teniendo esto en cuenta, y considerando que no siempre es necesario el mecanismo de sincronización en tiempo real, por ejemplo para datos que sabemos que no van a cambiar o que no lo van a hacer frecuentemente, la API de Firebase nos ofrece una forma alternativa de asociar el listener a una referencia de la base de datos de forma que éste sólo se ejecutará una vez, es decir, recibiremos el valor inicial del nodo en el momento de la suscripción a su referencia y ya no volveremos a recibir más actualizaciones de ese nodo. Esto nos evitará, en el caso de datos que no cambian, el suscribirnos a una referencia y tener que eliminar el listener inmediatamente después de recuperar su valor inicial. Para conseguir esto, el procedimiento sería análogo al ya mencionado con la diferencia de que utilizaríamos el método `addListenerForSingleValueEvent()`, en vez del ya mencionado `addValueEventListener()`.

## Filtrado y ordenación

Lo primero que debemos tener en cuenta es que en Firebase no vamos a tener todas las facilidades de ordenación y filtrado que suelen encontrarse en bases de datos SQL tradicionales. Por ejemplo, sólo podremos ordenar por un solo criterio, siempre ascendente, y que podrá basarse en la *clave* o el *valor* de los elementos de la lista, y adicionalmente el criterio de filtrado (si se utiliza) será dependiente del criterio de ordenación elegido. Así, si por ejemplo ordenamos una lista por el *valor* de sus elementos, si queremos utilizar algún criterio de filtrado éste se tendrá que basar también, necesariamente, en el *valor* de los elementos (no podría por ejemplo filtrar por la *clave*).

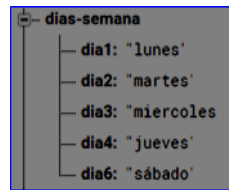
Empecemos por los diferentes métodos de ordenación disponibles. Firebase ofrece tres criterios de ordenación diferentes:

- `orderByKey()`. Ordena la información por la *clave* de cada elemento de la lista, en orden ascendente.
- `orderByValue()`. Ordena la información por el *valor* de cada elemento de la lista, en orden ascendente.
- `orderByChild()`. Ordena la información por el *valor* de una *clave hija* concreta de cada elemento de la lista, en orden ascendente.

La ordenación por clave (`orderByKey`) no necesita más explicación. En cuanto a las dos restantes, la ordenación directa por valor (`orderByValue`) tendrá más sentido cuando los elementos de la lista tengan un valor simple (numérico, alfanumérico o booleano), y la ordenación por el valor de una clave hija (`orderByChild`) nos será más útil cuando los elementos de la lista sean *objetos*, es decir, cuando contengan subelementos, y queramos ordenar por el valor de alguno de estos subelementos.



Veamos un par de ejemplos. Como ejemplo de lista con elementos simples tenemos una lista de días de la semana:

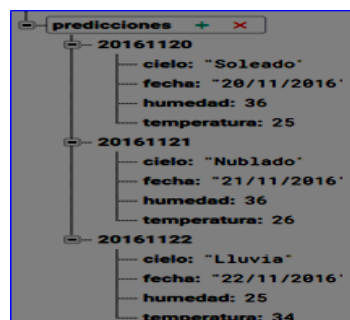


Si recuperamos esta lista ordenada por clave (`orderByKey`) obtendremos los elementos exactamente en el mismo orden que vemos en la imagen anterior, ya que en mi caso utilicé claves correlativas y ascendentes (aunque por supuesto esto no tiene por qué ser así necesariamente).

Si ordenamos la lista por valores (`orderByValue`) obtendríamos los elementos en este orden (por orden alfabético del campo valor):

```
{dia4: "jueves"}
{dia1: "lunes"}
{dia2: "martes"}
{dia3: "miercoles"}
{dia6: "sábado"}
```

Como ejemplo de una ordenación por clave hija suponiendo el siguiente ejemplo:



En este caso, podríamos ordenar la lista por el valor de cualquiera de las claves de los nodos hijo de cada elemento. Por ejemplo, si ordenamos por el campo “cielo” de cada elemento obtendríamos la lista en este orden:

```
{20161122: {cielo: "Lluvia", ...}}
{20161121: {cielo: "Nublado", ...}}
{20161120: {cielo: "Soleado", ...}}
```

Habiendo entendido los diferentes métodos de ordenación ya solo nos queda saber cómo aplicarlos al recuperar los datos de Firebase. Para ello, basta con utilizar el método correspondiente al crear la referencia a la base de datos, teniendo en cuenta además que el resultado ya no será un objeto de tipo `DatabaseReference`, sino de tipo `Query` (que realmente es una superclase de `DatabaseReference`).

Así, por ejemplo, para ordenar la lista de días de la semana por *clave* haríamos lo siguiente:



```
Query diasSemanaPorClave =  
    FirebaseDatabase.getInstance().getReference()  
        .child("dias-semana")  
        .orderByKey();
```

Para obtenerla ordenada por *valor* se haría de forma análoga:

```
Query diasSemanaPorValor =  
    FirebaseDatabase.getInstance().getReference()  
        .child("dias-semana")  
        .orderByValue();
```

Por último, para ordenar la lista de predicciones meteorológicas por la *clave hija* “cielo” de cada elemento, haríamos lo siguiente:

```
Query prediccionesPorClaveHija =  
    FirebaseDatabase.getInstance().getReference()  
        .child("predicciones")  
        .orderByChild("cielo");
```

Vamos ya con los métodos de filtrado o de consulta (query) en Firebase. Como hemos dicho anteriormente, el método de filtrado que apliquemos debe basarse necesariamente en el mismo campo por el que hayamos realizado la ordenación de la lista. Así, si ordenamos por clave, podremos filtrar por dicha clave. Si ordenamos por valor podremos filtrar por valor, y de forma análoga en el caso de ordenar por una clave hija.

Los distintos métodos de filtrado/consulta que tendremos disponibles serán los siguientes:

- `limitToFirst(N)`. La consulta sólo devolverá los primeros *N* elementos de la lista ordenada.
- `limitToLast(N)`. La consulta sólo devolverá los últimos *N* elementos de la lista ordenada.
- `startAt(...)`. La consulta sólo devolverá los elementos cuya clave/valor/valor\_clave\_hija (según el método de ordenación elegido) sea igual o superior al dato pasado como parámetro.
- `endAt(...)`. La consulta sólo devolverá los elementos cuya clave/valor/valor\_clave\_hija (según el método de ordenación elegido) sea igual o inferior al dato pasado como parámetro.
- `equalTo(...)`. La consulta sólo devolverá los elementos cuya clave/valor/valor\_clave\_hija (según el método de ordenación elegido) sea igual al dato pasado como parámetro.

Al contrario de lo que ocurre con los métodos de ordenación, en este caso sí podremos utilizar varios criterios de filtrado en la misma consulta, es decir, podremos combinar varios de los métodos anteriores para obtener sólo el rango de elementos necesario. Así, si ordenamos por ejemplo por valor, podríamos recuperar los primeros 50 valores de la lista (`limitToFirst(50)`) que sean mayores a un determinado valor “X” (`startAt(“X”)`). Este tipo de filtros podrían servirnos por ejemplo para paginar una lista grande de elementos.

La forma de aplicar los criterios de filtrado es análoga a la de los métodos de ordenación,



utilizando el/los métodos necesarios al crear la referencia a la base de datos. Así, por ejemplo, si volvemos a utilizar el ejemplo de los días de la semana, podríamos obtener los 2 días siguientes al “miércoles” (inclusive) de la siguiente forma:

```
Query diasSemanaPorValorFiltrado =  
    FirebaseDatabase.getInstance().getReference()  
        .child("dias-semana")  
        .orderByValue()  
        .startAt("miercoles")  
        .limitToFirst(2);
```

Con este criterio obtendríamos la siguiente lista:

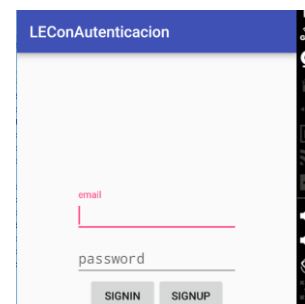
```
{dia3: "miercoles"}  
{dia6: "sábado"}
```

## Crear un proyecto con Autenticación

Vamos a empezar creando un proyecto nuevo con una actividad, podemos codificar un layout que nos permita introducir el usuario y la contraseña para poder logearnos. Una vez creado, nos vamos a Firebase de Google y realizamos los mismos pasos que habíamos hecho para el proyecto anterior.







Necesitaremos añadir el acceso de usuario en las dependencias de la app y el de la Base de Datos, importante que sean los dos 10.0.1:

```
implementation 'com.google.firebase:firebase-database:19.2.0' //Conexión a database  
implementation 'com.google.firebase:firebase-auth:19.2.0' //Autenticación de usuarios
```



Si queremos autenticarnos como usuarios, tenemos varias opciones: email/password, google, Facebook, twitter, github, anonymous. Nosotros vamos a ver el ejemplo de email/password y el de anonymous, en el resto de autenticaciones podéis seguir la documentación de FireBase, que en este caso y extrañamente está muy clara ;)

El primer paso que haremos, será activar los proveedores con los que queremos iniciar sesión. Nos vamos a la opción Auth y dentro de este a la pestaña Métodos de inicio de sesión. Tendremos que seleccionar ambos casos y habilitar el estado.

USUARIOS		MÉTODO DE INICIO DE SESIÓN	PLANTILLAS DE CORREO ELECTRÓNICO
Proveedores de inicio de sesión			
Proveedor		Estado	
 Correo electrónico/contraseña		Habilitado	
 Google		Inhabilitado	
 Facebook		Inhabilitado	
 Twitter		Inhabilitado	
 GitHub		Inhabilitado	
 Anónimo		Habilitado	



Una vez añadido los proveedores, tendremos dos formas de crear los usuarios: a través de la consola y desde la aplicación. Desde la consola nos tendremos que ir a la opción Auth a la pestaña usuarios, aquí podremos crear los usuarios que necesitemos. Los usuarios anónimos aparecerán solamente con el id único que asigna Firebase a cada usuario creado.

Authentication <span>CONFIGURACIÓN WEB</span>				
USUARIOS MÉTODO DE INICIO DE SESIÓN PLANTILLAS DE CORREO ELECTRÓNICO				
<div> <div> <div></div> <div>Busca usuarios introduciendo la dirección de correo electrónico exacta (correo@dominio.com) o su ...</div> </div> <div> <div>AÑADIR USUARIO</div> <div></div> </div> </div>				
Correo electrónico	Proveedores	Fecha de creación	Inicio de sesión	UID de usuario
xusa11@hotmailo.es		4 ago. 2016	5 ago. 2016	IzUolkpWdoWFIBN4TGsUTLl9WSF3
—	—	—	—	YISlnCypGZS0NoSkYewh57xcjND2
xusa01@hotmailo.es		4 ago. 2016	4 ago. 2016	r7L02yWvQpNIGZKEAc9ugSgCK...
<div> <div>Muestra hasta 500 usuarios</div> <div>Filas por página: 50 1-3 de 3</div> </div>				

```

El (
un
en
oc
pub
    ///Escuchador de cambios en los usuarios
    mAuthListener = new FirebaseAuth.AuthStateListener() {
        @Override
        public void onAuthStateChanged(@NonNull FirebaseAuth firebaseAuth) {
            FirebaseUser user = firebaseAuth.getCurrentUser();
            if (user != null) {
                Toast.makeText(MainActivity.this, user.getEmail() + " LOGEADO", Toast.LENGTH_SHORT).show();
                mAuth = firebaseAuth;
            } else {
                Toast.makeText(MainActivity.this, "Usuario NULO", Toast.LENGTH_SHORT).show();
            }
        }
    };
    FirebaseAuth.AuthStateListener mAuthListener;
    FirebaseAuth mAuth;
    FirebaseDatabase database;

    @Override
    public void onStart() {
        super.onStart();
        mAuth.addAuthStateListener(mAuthListener);
    }

    @Override
    public void onStop() {
        super.onStop();
        if (mAuthListener != null) {
            mAuth.removeAuthStateListener(mAuthListener);
            mAuth.signOut();
        }
    }
}

```

**El escuchador AuthStateListener deberá ser incluido en el método onCreate de la aplicación.**

El siguiente paso será decidir si queremos permitir crear usuarios nuevos, o solamente logearnos con alguno existente. En nuestra aplicación vamos a hacer las dos cosas, he incluso nos logearemos como anónimos. El código aparece a continuación, y un enlace con toda la información. <https://firebase.google.com/docs/auth/android/password-auth>



```

//////// Crear usuario nuevo y iniciar sesión
btCrear.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        mAuth.createUserWithEmailAndPassword(user.getText().toString(), password.getText().toString())
            .addOnCompleteListener(MainActivity.this, new OnCompleteListener<AuthResult>() {
                @Override
                public void onComplete(@NonNull Task<AuthResult> task) {
                    if (task.isSuccessful()) {
                        Toast.makeText(MainActivity.this, "Usuario creado", Toast.LENGTH_SHORT).show();
                        iniciarAplicacion(task.getResult().getUser().getEmail().split("@")[0]);
                    } else Toast.makeText(MainActivity.this, "Problemas al crear usuario" + task.getExceptioni
                }
            });
    }
});

////////Iniciar sesión con usuario y contraseña
btIniciar.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        mAuth.signInWithEmailAndPassword(user.getText().toString(), password.getText().toString())
            .addOnCompleteListener(MainActivity.this, (task) -> {
                if (!task.isSuccessful()) {
                    Toast.makeText(MainActivity.this, "Authentication failed:" + task.getException(), Toast.LEN
                }
                else iniciarAplicacion(task.getResult().getUser().getEmail().split("@")[0]);
            });
    }
});

///////// Iniciar sesión con anónimo
btAnonimus.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        mAuth.signInAnonymously().addOnCompleteListener(MainActivity.this, new OnCompleteListener<AuthResult>() {
            @Override
            public void onComplete(@NonNull Task<AuthResult> task) {
                if (!task.isSuccessful()) {
                    Toast.makeText(MainActivity.this, "Authentication failed:" + task.getException(), Toast.LENGTH_SHOF
                }
                else iniciarAplicacion(task.getResult().getUser().getEmail().split("@")[0]);
            }
        });
    }
});

```

Como podemos ver en el código, lo que se hace es añadir el mismo escuchador a cualquiera de los métodos que hayamos elegido (login, anónimo o añadir usuario) sobre un objeto FirebaseAuth. Con estos pasos tendremos la autenticación controlada, y podremos seguir con nuestra aplicación.

Como el almacenamiento de datos en Firebase, se hace de la misma forma que vimos en el ejemplo sin autenticación, no repetiremos el proceso. A partir del siguiente enlace, podemos ver la información necesaria para guardar datos y leer datos de Firebase en Android.

<https://firebase.google.com/docs/database/android/save-data>

## Firestore y RecyclerView

Firestore nos permite asociar fácilmente a un control ListView o RecyclerView, una referencia a una lista de elementos de una base de datos Firestore. De esta forma, el control se actualizará automáticamente cada vez que se produzca cualquier cambio en la lista, sin



tener que gestionar manualmente por nuestra parte los eventos de la lista, ni tener que almacenar en una estructura paralela la información, ni tener que construir gran parte de los adaptadores necesarios... en resumen, ahorrando muchas líneas de código y evitando muchos posibles errores.

Para utilizar FirebaseUI lo primero que tendremos que hacer será añadir la referencia a la librería en el fichero *build.gradle* de nuestro módulo principal. Hay que tener en cuenta que cada versión de FirebaseUI es compatible únicamente con una versión concreta de Firebase, por lo que debemos asegurar que las versiones utilizadas de ambas librerías son coherentes. En la página de FirebaseUI tenéis disponible la tabla de compatibilidades entre versiones. <https://github.com/firebase/FirebaseUI-Android>

En mi caso:

```
implementation 'com.google.firebase:firebase-database:19.2.0'//Conexión a database
implementation 'com.firebaseui:firebase-ui-database:6.1.0'
```

Esta librería nos provee de un Adaptador derivado de *RecyclerView.ViewHolder* que gestionará automáticamente la carga de los datos que le pasemos como referencia en la vista asignada por el Holder, para ello lo primero que deberemos crear es el *ViewHolder* personalizado que gestione los datos de nuestra aplicación (retomamos el ejemplo de *LeeConAutenticación*).

```
public class HolderItem extends RecyclerView.ViewHolder implements View.OnClickListener {
    private TextView usuario, mensaje;
    private View.OnClickListener listener;

    public HolderItem(View v)
    {
        super(v);
        v.setOnClickListener(this);
        usuario=(TextView)v.findViewById(R.id.usuario);
        mensaje=(TextView)v.findViewById(R.id.mensaje);
    }

    void bind(Items item)
    {
        usuario.setText(item.user );
        mensaje.setText(item.msg);
    }

    void onClickListener(View.OnClickListener listener) { this.listener=listener; }
    @Override
    public void onClick(View view) { if(listener!=null ) listener.onClick(view); }
}
```

Como podemos ver, el código es el típico del Holder que ya conocemos, al que le hemos añadido la implementación de un listener para el evento *onClick* de forma que al pulsar sobre un elemento del recycler podamos acceder después a ese elemento.

El siguiente paso será crear el adaptador para nuestro *RecyclerView*. FirebaseUI nos facilita este paso proporcionándonos una clase genérica, llamada *FirebaseRecyclerViewAdapter*, que podemos utilizar con nuestro *ViewHolder* personalizado que acabamos de crear y la clase java que utilicemos para encapsular la información de cada elemento de la lista. Al crear un objeto de esta clase, debemos pasarle un objeto de la misma librería, de tipo *FirebaseRecyclerViewOptions*.



```

FirebaseRecyclerOptions<Items> firebaseRecyclerOptions=new FirebaseRecyclerOptions.Builder<Items>()
    .setQuery(query,Items.class).build();

```

Al que debemos pasarle la siguiente información:

- El objeto clase de nuestro Item (Items.class),
- La referencia al nodo de la base de datos que contiene la lista de elementos que queremos mostrar en el control.

La forma más limpia de implementar el código sería creando una clase derivada de FirebaseRecyclerAdapter, en donde sobrescribiríamos, obligatoriamente, los dos métodos y el constructor. He implementaríamos los listener necesarios para el funcionamiento de los eventos.

```

public class Adapter extends FirebaseRecyclerAdapter<Items, HolderItem> implements View.OnClickListener {

    private View.OnClickListener listener;

    Adapter(@NonNull FirebaseRecyclerOptions<Items> options) {
        super(options);
    }

    @Override
    protected void onBindViewHolder(@NonNull HolderItem holder, int position, @NonNull Items model) {
        holder.bind(model);
    }

    @Override
    public HolderItem onCreateViewHolder(ViewGroup parent, int viewType) {
        View view = LayoutInflater.from(parent.getContext())
            .inflate(R.layout.linearrecycler, parent, false);
        view.setOnClickListener(this);

        return new HolderItem(view);
    }

    void onClickListener(View.OnClickListener listener)
    {
        this.listener=listener;
    }

    @Override
    public void onClick(View v) {
        if(listener!=null) listener.onClick(v);
    }
}

```

Ahora solo quedaría crear un objeto del tipo adaptador que nos hemos creado, pasándole el elemento firebaseRecyclerOption y ya tendríamos casi todo hecho.

```

FirebaseRecyclerOptions<Items> firebaseRecyclerOptions=new FirebaseRecyclerOptions.Builder<Items>()
    .setQuery(query,Items.class).build();

recyclerView=(RecyclerView) findViewById(R.id.recycler);
adapter=new Adapter(firebaseRecyclerOptions);
//Click para eliminar elemento (se detecta en el holder)
adapter.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Toast.makeText(Application.this,"Elemento eliminado" +recyclerView.getChildAdapterPosition(view), To
        String key=adapter.getRef(recyclerView.getChildAdapterPosition(view)).getKey();
        database.child(key).removeValue();
    }
});
recyclerView.setAdapter(adapter);
recyclerView.setLayoutManager(new LinearLayoutManager(this));

```

No deberemos olvidar iniciar el escuchador del adaptador al comenzar la aplicación y cerrarlo al acabar:

```

@Override
protected void onStart() {
    super.onStart();
    adapter.startListening();
}

@Override
protected void onStop() {
    adapter.stopListening();
    super.onStop();
}

```

 **Ejercicio propuesto FireBaseConAutenticacion**





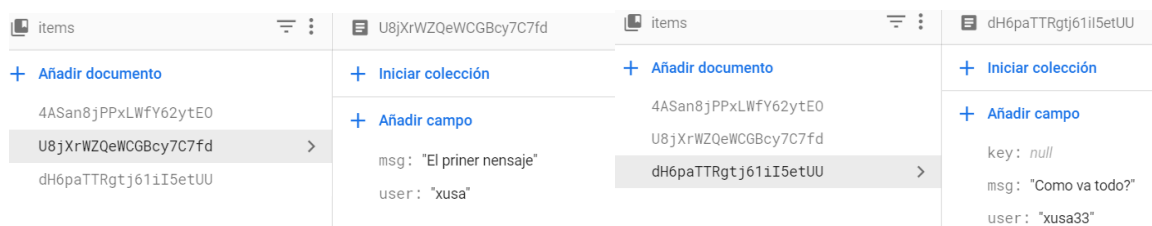
## Cloud Firestore

Base de datos NoSQL flexible, escalable y en la nube a fin de almacenar y sincronizar datos para la programación en el lado del cliente y del servidor. Al igual que Firebase Realtime Database, mantiene los datos sincronizados entre apps cliente a través de agentes de escucha en tiempo real. El modelo de datos de Cloud Firestore admite estructuras de datos flexibles y jerárquicas. Almacena tus datos en documentos organizados en colecciones, los documentos pueden contener objetos anidados complejos, además de subcolecciones. Cloud Firestore está optimizado para almacenar grandes colecciones de documentos pequeños.

El modelo de datos está formado por documentos, cada documento contiene un conjunto de pares clave-valor. Todos los documentos se deben almacenar en colecciones, los documentos pueden contener subcolecciones y objetos anidados, y ambos pueden incluir campos primitivos como strings o tipos de objetos complejos como listas.

Las colecciones y los documentos se crean de manera implícita en Cloud Firestore. Solo debes asignar datos a un documento dentro de una colección. Si la colección o el documento no existen, Cloud Firestore los crea.

Cada documento está identificado por una clave única, que puede generarse automáticamente o que se puede añadir a la vez que el documento:



Son muy similares a JSON, de hecho, básicamente son JSON. Existen algunas diferencias, por ejemplo: **los documentos admiten tipos de datos adicionales** y su tamaño se limita a 1 MB, pero en general, puedes tratar los documentos como registros JSON livianos.

Los documentos viven en colecciones, que simplemente son contenedores de documentos. Por ejemplo, podrías tener una colección llamada users con los distintos usuarios de tu app, en la que haya un documento que represente a cada uno:



Vamos a modificar el ejercicio propuesto FireBaseConAutenticacion, pero ahora para Cloud Firestore. Copiamos en proyecto en otra carpeta y pasamos a realizar los cambios oportunos dentro de Aplicación.java. Primero deberemos añadir la dependencia:

```
implementation 'com.google.firebase:firebase-firestore:21.2.1'
```

En nuestro proyecto de Firebase LeeConAutenticacion, añadiremos una base de datos más, pero ahora de Cloud Firestore (pueden convivir las dos BD en un mismo proyecto, y referenciar a la que necesitemos en cada caso). Crearemos de forma manual la colección 'items' y añadiremos un documento a esta, como hicimos para el anterior caso (tendrá user y msg como datos ).

En esta ocasión deberemos crearnos un objeto de tip FirebaseFirestore y hacer referencia a la collection items,

```
db = FirebaseFirestore.getInstance().collection("items");
```

En el botón para añadir un item a la colección, ahora deberemos añadir el siguiente código:

```
btMandar.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        if(!etMensaje.getText().toString().isEmpty())
        {
            db.add(new Items(etMensaje.getText().toString(), user)).addOnSuccessListener(new OnSuccessListener<DocumentReference>() {
                @Override
                public void onSuccess(DocumentReference documentReference) {
                    Toast.makeText(Aplicacion.this, "Elemento añadido", Toast.LENGTH_SHORT).show();
                }
            })
            .addOnFailureListener((e) -> {
                Toast.makeText(Aplicacion.this, "Error adding document", Toast.LENGTH_SHORT).show();
            });
        }
    }
});
```

Como se puede ver en el código anterior, a diferencia de con DataBase, al método de añadir elemento se le asignará un escuchador para comprobar si el proceso se ha solucionado con éxito.

El botón buscar podría ser algo parecido a lo siguiente:

```
//Hacer búsqueda según condición
btBuscar.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {

        if(etUser.getText().toString().isEmpty()) cargarAdaptador(db);
        else{
            Query query = db.whereEqualTo("user",etUser.getText().toString());
            cargarAdaptador(query);
        }
    }
});
```



Con el método cargarAdaptador de la siguiente manera:

```
private void cargarAdaptador(final Query query) {
    query.get().addOnCompleteListener(new OnCompleteListener<QuerySnapshot>() {
        @Override
        public void onComplete(@NonNull Task<QuerySnapshot> task) {
            if (task.isSuccessful()) cargarRecycler(query);
            else {
                Toast.makeText(Aplicacion.this, "Error getting document", Toast.LENGTH_SHORT).show();
            }
        }
    });
}
```

Que a su vez llamará a cargarRecycler:

```
private void cargarRecycler(Query query) {
    FirestoreRecyclerOptions<Items> firestoreRecyclerOptions = new FirestoreRecyclerOptions.Builder<Items>()
        .setQuery(query, Items.class).build();
    recyclerView = (RecyclerView) findViewById(R.id.recycler);
    adapter = new Adapter(firestoreRecyclerOptions);
    adapter.startListening();
    //Click para eliminar elemento (se detecta en el holder)
    adapter.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Toast.makeText(Aplicacion.this, "Elemento eliminado" + recyclerView.getChildAdapterPosition(view), Toast.LENGTH_SHORT).show();
            adapter.getSnapshots().getSnapshot(recyclerView.getChildAdapterPosition(view)).getReference().delete();
        }
    });
    recyclerView.setAdapter(adapter);
    recyclerView.setLayoutManager(new LinearLayoutManager(this));
}
```

No olvidar añadir la implementación para el recyclerui

```
implementation 'com.firebaseui:firebase-ui-firestore:6.1.0'
```

Para más información <https://firebase.google.com/docs/firestore>

*Instalar servicios de google en Genimotion.*

Para poder probar las APP en Genimotion, necesitaremos instalar los servicios de Google para ello. En el siguiente enlace podemos ver cuál es el proceso <https://z3ntu.github.io/2015/12/10/play-services-with-genymotion.html>

